Public Key Cryptography Implementation Report

CPE-321 Introduction to Computer Security

Tasks Implemented:

Task 1: Diffie-Hellman Key Exchange

Task 2: MITM Key Fixing & Negotiated Groups

Task 3: Textbook RSA & MITM via Malleability

Tools: Python 3, PyCryptodome. AI coding assistant (Claude) used for implementation.

Task 1: Diffie-Hellman Key Exchange

This program implements the Diffie-Hellman key exchange protocol. Alice and Bob agree on public parameters (q, a), each pick a random private key, compute public values, exchange them, and derive a shared secret. The shared secret is hashed with SHA-256 (truncated to 16 bytes) to produce an AES-128 key. Both parties then encrypt and decrypt messages using AES-CBC.

The protocol is tested first with small parameters (q=37, a=5) and then with the IETF-recommended 1024-bit parameters.

Source Code: task1_diffie_hellman.py

```
#!/usr/bin/env python3
"""
Task 1: Diffie-Hellman Key Exchange Implementation
===================================================
Emulates the full DH key exchange protocol between Alice and Bob.
- First with small parameters (q=37, a=5)
- Then with IETF 1024-bit parameters
"""

import secrets
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Util.Padding import pad, unpad


def int_to_bytes(n: int) -> bytes:
    """Convert a non-negative integer to a big-endian byte string."""
    if n == 0:
        return b'\x00'
    length = (n.bit_length() + 7) // 8
    return n.to_bytes(length, byteorder='big')


def derive_key(shared_secret: int) -> bytes:
    """Derive a 16-byte AES key from the shared secret using SHA-256."""
    h = SHA256.new()
    h.update(int_to_bytes(shared_secret))
    return h.digest()[:16]  # truncate to 16 bytes for AES-128


def aes_cbc_encrypt(key: bytes, iv: bytes, plaintext: bytes) -> bytes:
    """Encrypt plaintext with AES-CBC, applying PKCS7 padding."""
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.encrypt(pad(plaintext, AES.block_size))


def aes_cbc_decrypt(key: bytes, iv: bytes, ciphertext: bytes) -> bytes:
    """Decrypt ciphertext with AES-CBC, removing PKCS7 padding."""
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size)


def diffie_hellman_exchange(q: int, a: int, label: str = ""):
    """
    Run the full Diffie-Hellman key exchange between Alice and Bob.

    Parameters
    ----------
    q : int  -- the prime modulus
    a : int  -- the generator
    label : str -- descriptive label for output
    """
    print("=" * 70)
    print(f"  Diffie-Hellman Key Exchange -- {label}")
    print("=" * 70)
    # --- Public parameters ---
    print(f"\nPublic parameters:")
    print(f"  q = {q}")
    print(f"  a = {a}")

    # --- Alice picks a random private key X_A in {2, ..., q-2} ---
    X_A = secrets.randbelow(q - 2) + 2  # range [2, q-1]
    Y_A = pow(a, X_A, q)

    print(f"\nAlice:")
    print(f"  Private key  X_A = {X_A}")
    print(f"  Public value Y_A = a^X_A mod q = {Y_A}")

    # --- Bob picks a random private key X_B in {2, ..., q-2} ---
    X_B = secrets.randbelow(q - 2) + 2
    Y_B = pow(a, X_B, q)

    print(f"\nBob:")
    print(f"  Private key  X_B = {X_B}")
    print(f"  Public value Y_B = a^X_B mod q = {Y_B}")

    # --- Exchange public values and compute shared secret ---
    # Alice sends Y_A to Bob; Bob sends Y_B to Alice.
    s_alice = pow(Y_B, X_A, q)
    s_bob   = pow(Y_A, X_B, q)

    print(f"\nShared secret computation:")
    print(f"  Alice computes s = Y_B^X_A mod q = {s_alice}")
    print(f"  Bob   computes s = Y_A^X_B mod q = {s_bob}")
    assert s_alice == s_bob, "ERROR: shared secrets do not match!"
    print(f"  [ok] Shared secrets match: s = {s_alice}")

    # --- Derive symmetric key ---
    k_alice = derive_key(s_alice)
    k_bob   = derive_key(s_bob)

    print(f"\nDerived AES-128 key (SHA-256 truncated to 16 bytes):")
    print(f"  Alice: k = {k_alice.hex()}")
    print(f"  Bob:   k = {k_bob.hex()}")
    assert k_alice == k_bob, "ERROR: derived keys do not match!"
    print(f"  [ok] Keys match")

    # --- Encrypted message exchange ---
    # Use a shared initialization vector (16 zero bytes for simplicity)
    iv = b'\x00' * 16

    # Alice encrypts m0 = "Hi Bob!" and sends c0 to Bob
    m0 = b"Hi Bob!"
    c0 = aes_cbc_encrypt(k_alice, iv, m0)
    print(f"\nAlice -> Bob:")
    print(f"  Plaintext  m0 = {m0.decode()}")
    print(f"  Ciphertext c0 = {c0.hex()}")

    # Bob decrypts c0
    m0_dec = aes_cbc_decrypt(k_bob, iv, c0)
    print(f"  Bob decrypts:  {m0_dec.decode()}")
    assert m0_dec == m0, "ERROR: Bob failed to decrypt Alice's message!"
    print(f"  [ok] Bob successfully decrypted Alice's message")

    # Bob encrypts m1 = "Hi Alice!" and sends c1 to Alice
    m1 = b"Hi Alice!"
    c1 = aes_cbc_encrypt(k_bob, iv, m1)
    print(f"\nBob -> Alice:")
    print(f"  Plaintext  m1 = {m1.decode()}")
    print(f"  Ciphertext c1 = {c1.hex()}")

    # Alice decrypts c1
    m1_dec = aes_cbc_decrypt(k_alice, iv, c1)
    print(f"  Alice decrypts: {m1_dec.decode()}")
    assert m1_dec == m1, "ERROR: Alice failed to decrypt Bob's message!"
    print(f"  [ok] Alice successfully decrypted Bob's message")

    print()
```

```
# ============================================================
# Main -- run with both parameter sets
# ============================================================
if __name__ == "__main__":

    # ----------------------------------------------------------
    # Part A: Small group  q = 37, a = 5
    # ----------------------------------------------------------
    diffie_hellman_exchange(q=37, a=5, label="Small group (q=37, a=5)")

    # ----------------------------------------------------------
    # Part B: IETF 1024-bit parameters
    # ----------------------------------------------------------
    q_ietf = int(
        "B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C6"
        "9A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C0"
        "13ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
        "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0"
        "A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708"
        "DF1FB2BC2E4A4371",
        16
    )

    a_ietf = int(
        "A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507F"
        "D6406CFF14266D31266FEA1E5C41564B777E690F5504F213"
        "160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
        "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28A"
        "D662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24"
        "855E6EEB22B3B2E5",
        16
    )

    diffie_hellman_exchange(q=q_ietf, a=a_ietf,
                            label="IETF 1024-bit parameters")
```

Output

```
======================================================================
  Diffie-Hellman Key Exchange -- Small group (q=37, a=5)
======================================================================

Public parameters:
  q = 37
  a = 5

Alice:
  Private key  X_A = 15
  Public value Y_A = a^X_A mod q = 29

Bob:
  Private key  X_B = 34
  Public value Y_B = a^X_B mod q = 3

Shared secret computation:
  Alice computes s = Y_B^X_A mod q = 11
  Bob   computes s = Y_A^X_B mod q = 11
  [ok] Shared secrets match: s = 11

Derived AES-128 key (SHA-256 truncated to 16 bytes):
  Alice: k = e7cf46a078fed4fafd0b5e3aff144802
  Bob:   k = e7cf46a078fed4fafd0b5e3aff144802
  [ok] Keys match

Alice -> Bob:
  Plaintext  m0 = Hi Bob!
  Ciphertext c0 = d16b6ee2d3d6ebc916d26861d838a714
  Bob decrypts:  Hi Bob!
  [ok] Bob successfully decrypted Alice's message

Bob -> Alice:
  Plaintext  m1 = Hi Alice!
  Ciphertext c1 = 46020add586c36683391e972771ee711
  Alice decrypts: Hi Alice!
  [ok] Alice successfully decrypted Bob's message

======================================================================
  Diffie-Hellman Key Exchange -- IETF 1024-bit parameters
======================================================================

Public parameters:
  q = 1243253391468893845404940910854566300098568827418728061817312790184918208001194600223674037697950082500211917675834232214...
  a = 1157402005271091642395234147609261555344857158600902615321541073139462184591494023751781794580414617237232315638393162515...

Alice:
  Private key  X_A = 3351890427849581528470019994080135427287015651111410217087969115265760572473015454043554943910783772418092...
  Public value Y_A = a^X_A mod q = 6848168243070943341463890366859903345656181133004737993609148395435389409482227916801210184 6...
```

Bob:
  Private key X_B = 7655015255799113635629091760177101138884073882752177019644442901893435348471352529866863260459358042142563...
  Public value Y_B = a^X_B mod q = 2217578970860848058094570597823629670195995809890643650615219556936199259408975487193004605 2...

Shared secret computation:
  Alice computes s = Y_B^X_A mod q = 10411875543754451771178836123076404567955626142276957255796247261529026434961524786212037 7...
  Bob   computes s = Y_A^X_B mod q = 10411875543754451771178836123076404567955626142276957255796247261529026434961524786212037 7...
  [ok] Shared secrets match: s = 1041187554375445177117883612307640456795562614227695725579624726152902643496152478621203773073...

Derived AES-128 key (SHA-256 truncated to 16 bytes):
  Alice: k = 6649d00af493ad33be4c3e895455c548
  Bob: k = 6649d00af493ad33be4c3e895455c548
  [ok] Keys match

Alice -> Bob:
  Plaintext m0 = Hi Bob!
  Ciphertext c0 = 051c24bae4c9d5109c34c5b48b6746da
  Bob decrypts: Hi Bob!
  [ok] Bob successfully decrypted Alice's message

Bob -> Alice:
  Plaintext m1 = Hi Alice!
  Ciphertext c1 = ebcbf830519636841c172f85f394d427
  Alice decrypts: Hi Alice!
  [ok] Alice successfully decrypted Bob's message
```
```

Task 2: MITM Key Fixing & Negotiated Groups

This program demonstrates two man-in-the-middle attacks on Diffie-Hellman.

Part 1 (Key Fixing): Mallory intercepts the exchange and replaces both public values Y_A and Y_B with q. Since q^X mod q = 0 for any X, both Alice and Bob compute s = 0. Mallory knows s = 0 and can derive the same symmetric key to decrypt all messages.

Part 2 (Generator Tampering): Mallory tampers with the generator a:
- a = 1: All public values become 1, so s = 1.
- a = q: All public values become 0 (q mod q), so s = 0.
- a = q-1: Public values are in {1, q-1}, so s is in {1, q-1}. Mallory tries both candidates.

Source Code: task2_mitm_attack.py

```
#!/usr/bin/env python3
"""
Task 2: MITM Key Fixing & Negotiated Groups
=============================================
Demonstrates two attacks on Diffie-Hellman:
  Part 1 -- Mallory replaces Y_A and Y_B with q  (key fixing)
  Part 2 -- Mallory tampers with the generator a  (negotiated groups)
"""

import secrets
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Util.Padding import pad, unpad


# ?? Helpers (same as Task 1) ?????????????????????????????????????????

def int_to_bytes(n: int) -> bytes:
    """Convert a non-negative integer to a big-endian byte string."""
    if n == 0:
        return b'\x00'
    length = (n.bit_length() + 7) // 8
    return n.to_bytes(length, byteorder='big')


def derive_key(shared_secret: int) -> bytes:
    """Derive a 16-byte AES key from the shared secret using SHA-256."""
    h = SHA256.new()
    h.update(int_to_bytes(shared_secret))
    return h.digest()[:16]


def aes_cbc_encrypt(key: bytes, iv: bytes, plaintext: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.encrypt(pad(plaintext, AES.block_size))


def aes_cbc_decrypt(key: bytes, iv: bytes, ciphertext: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size)


# ?? IETF 1024-bit parameters ????????????????????????????????????????

Q = int(
    "B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C6"
    "9A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C0"
    "13ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
    "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0"
    "A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708"
    "DF1FB2BC2E4A4371",
    16,
)

A = int(
    "A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507F"
    "D6406CFF14266D31266FEA1E5C41564B777E690F5504F213"
    "160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
    "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28A"
    "D662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24"
    "855E6EEB22B3B2E5",
    16,
)

IV = b'\x00' * 16  # shared initialization vector


# =====================================================================
# Part 1 -- MITM Key Fixing: Mallory replaces Y_A -> q and Y_B -> q
# =====================================================================

def mitm_key_fixing(q: int, a: int):
    print("=" * 70)
    print("  Task 2 Part 1 -- MITM Key Fixing (Y_A -> q, Y_B -> q)")
    print("=" * 70)

    # ?? Alice generates her key pair normally ??
    X_A = secrets.randbelow(q - 2) + 2
    Y_A = pow(a, X_A, q)
    print(f"\nAlice:")
    print(f"  X_A = {X_A}")
    print(f"  Y_A = {Y_A}")

    # ?? Bob generates his key pair normally ??
    X_B = secrets.randbelow(q - 2) + 2
    Y_B = pow(a, X_B, q)
    print(f"\nBob:")
    print(f"  X_B = {X_B}")
    print(f"  Y_B = {Y_B}")

    # ?? Mallory intercepts and replaces both public values with q ??
    Y_A_to_bob = q    # Mallory sends q to Bob instead of Y_A
    Y_B_to_alice = q  # Mallory sends q to Alice instead of Y_B
    print(f"\nMallory intercepts:")
    print(f"  Replaces Y_A -> q (sends q to Bob)")
    print(f"  Replaces Y_B -> q (sends q to Alice)")

    # ?? Alice computes her shared secret using the tampered Y_B ??
    s_alice = pow(Y_B_to_alice, X_A, q)   # q^X_A mod q = 0
    k_alice = derive_key(s_alice)
    print(f"\nAlice computes:")
    print(f"  s = (received Y_B)^X_A mod q = q^X_A mod q = {s_alice}")
    print(f"  k = SHA256(s)[:16] = {k_alice.hex()}")

    # ?? Bob computes his shared secret using the tampered Y_A ??
    s_bob = pow(Y_A_to_bob, X_B, q)       # q^X_B mod q = 0
    k_bob = derive_key(s_bob)
    print(f"\nBob computes:")
    print(f"  s = (received Y_A)^X_B mod q = q^X_B mod q = {s_bob}")
    print(f"  k = SHA256(s)[:16] = {k_bob.hex()}")

    assert s_alice == s_bob == 0, "Expected s = 0 for both!"
    print(f"\n  [ok] Both shared secrets = 0 (as Mallory predicted)")

    # ?? Mallory knows s = 0, derives the same key ??
    s_mallory = 0
    k_mallory = derive_key(s_mallory)
    print(f"\nMallory knows s = 0:")
    print(f"  k_mallory = SHA256(0)[:16] = {k_mallory.hex()}")
    assert k_mallory == k_alice == k_bob
    print(f"  [ok] Mallory's key matches Alice's and Bob's key")

    # ?? Alice encrypts "Hi Bob!" -> c0 ??
    m0 = b"Hi Bob!"
    c0 = aes_cbc_encrypt(k_alice, IV, m0)
    print(f"\nAlice -> Bob:")
    print(f"  m0 = {m0.decode()}")
    print(f"  c0 = {c0.hex()}")

    # ?? Bob encrypts "Hi Alice!" -> c1 ??
    m1 = b"Hi Alice!"
    c1 = aes_cbc_encrypt(k_bob, IV, m1)
    print(f"\nBob -> Alice:")
    print(f"  m1 = {m1.decode()}")
    print(f"  c1 = {c1.hex()}")

    # ?? Mallory decrypts both ciphertexts ??
    m0_mallory = aes_cbc_decrypt(k_mallory, IV, c0)
    m1_mallory = aes_cbc_decrypt(k_mallory, IV, c1)
    print(f"\nMallory decrypts:")
    print(f"  c0 -> {m0_mallory.decode()}")
    print(f"  c1 -> {m1_mallory.decode()}")
    assert m0_mallory == m0 and m1_mallory == m1
    print(f"  [ok] Mallory successfully recovered both plaintext messages!")

    print()


# ======================================================================
# Part 2 -- Generator Tampering: Mallory replaces a with 1, q, or q-1
# ======================================================================

def mitm_generator_tamper(q: int, a_original: int, a_tampered: int, label: str):
    """
    Run DH with a tampered generator, then show Mallory can recover messages.
    """
    print("=" * 70)
    print(f"  Task 2 Part 2 -- Generator Tamper: a -> {label}")
    print("=" * 70)

    a = a_tampered  # Alice and Bob unknowingly use the tampered generator

    # ?? Alice generates her key pair ??
    X_A = secrets.randbelow(q - 2) + 2
    Y_A = pow(a, X_A, q)
    print(f"\nAlice (using tampered a):")
    print(f"  X_A = {X_A}")
    print(f"  Y_A = a^X_A mod q = {Y_A}")

    # ?? Bob generates his key pair ??
    X_B = secrets.randbelow(q - 2) + 2
    Y_B = pow(a, X_B, q)
    print(f"\nBob (using tampered a):")
    print(f"  X_B = {X_B}")
    print(f"  Y_B = a^X_B mod q = {Y_B}")

    # ?? Both compute shared secret normally ??
    s_alice = pow(Y_B, X_A, q)
    s_bob   = pow(Y_A, X_B, q)
    k_alice = derive_key(s_alice)
    k_bob   = derive_key(s_bob)

    print(f"\nShared secret:")
    print(f"  Alice: s = {s_alice}")
    print(f"  Bob:   s = {s_bob}")
    assert s_alice == s_bob, "Shared secrets should match"
    print(f"  [ok] Shared secrets match: s = {s_alice}")
    print(f"  k = {k_alice.hex()}")

    # ?? Alice encrypts m0, Bob encrypts m1 ??
    m0 = b"Hi Bob!"
    c0 = aes_cbc_encrypt(k_alice, IV, m0)
    m1 = b"Hi Alice!"
    c1 = aes_cbc_encrypt(k_bob, IV, m1)
    print(f"\nAlice -> Bob:  c0 = {c0.hex()}")
    print(f"Bob -> Alice:  c1 = {c1.hex()}")
    # ?? Mallory's attack: determine possible s values ??
    # Depending on the tampered generator:
    #   a = 1       -> Y = 1, s = 1
    #   a = q       -> Y = 0, s = 0
    #   a = q-1     -> Y in {1, q-1}, s in {1, q-1}
    if a_tampered == 1:
        candidates = [1]
        explanation = "a=1 ==> Y_A=Y_B=1 ==> s = 1^X mod q = 1"
    elif a_tampered == q:
        candidates = [0]
        explanation = "a=q ==> Y_A=Y_B=0 ==> s = 0^X mod q = 0"
    elif a_tampered == q - 1:
        candidates = [1, q - 1]
        explanation = ("a=q-1 ==> Y in {1, q-1} ==> s in {1, q-1}; "
                       "Mallory tries both")
    else:
        candidates = []
        explanation = "Unknown tampered value"

    print(f"\nMallory's reasoning:")
    print(f"  {explanation}")
    print(f"  Candidate shared secrets: {candidates}")

    # ?? Mallory brute-forces the (small) candidate set ??
    recovered_m0 = None
    recovered_m1 = None
    for s_guess in candidates:
        k_guess = derive_key(s_guess)
        try:
            recovered_m0 = aes_cbc_decrypt(k_guess, IV, c0)
            recovered_m1 = aes_cbc_decrypt(k_guess, IV, c1)
            print(f"\n  s = {s_guess} -> k = {k_guess.hex()}")
            print(f"    c0 decrypts to: {recovered_m0.decode()}")
            print(f"    c1 decrypts to: {recovered_m1.decode()}")
            print(f"    [ok] Mallory recovered both messages!")
            break
        except (ValueError, UnicodeDecodeError):
            print(f"  s = {s_guess} -> decryption failed (wrong key), trying next...")

    assert recovered_m0 == m0 and recovered_m1 == m1, \
        "Mallory failed to recover messages!"

    print()


# =====================================================================
# Main
# =====================================================================

if __name__ == "__main__":

    # ?? Part 1: Key Fixing Attack ??
    mitm_key_fixing(Q, A)

    # ?? Part 2: Generator Tampering ??
    # Case 1: a = 1
    mitm_generator_tamper(Q, A, a_tampered=1, label="1")

    # Case 2: a = q
    mitm_generator_tamper(Q, A, a_tampered=Q, label="q")

    # Case 3: a = q - 1
    mitm_generator_tamper(Q, A, a_tampered=Q - 1, label="q-1")
```

Output

```
======================================================================
  Task 2 Part 1 -- MITM Key Fixing (Y_A -> q, Y_B -> q)
======================================================================

Alice:
  X_A = 3393190849040616618458472951528184902731898629584140016327346630230202355300298487476892792916453691640770358151011665O...
  Y_A = 1581983866894300402876319742170564560324623573946981228129315918887406789265383123926824131565891272371635790401763239 3...
  
Bob:
  X_B = 4318991527367047870023117228787329710578049501747933915311345211729865678048098315914927926530296591062311072545819869 9...
  Y_B = 6844906491542365625167398853994049657834549415550346247897267849529798434407401356386162898871930862621388669787684555 2...

Mallory intercepts:
  Replaces Y_A -> q (sends q to Bob)
  Replaces Y_B -> q (sends q to Alice)

Alice computes:
  s = (received Y_B)^X_A mod q = q^X_A mod q = 0
  k = SHA256(s)[:16] = 6e340b9cffb37a989ca544e6bb780a2c

Bob computes:
  s = (received Y_A)^X_B mod q = q^X_B mod q = 0
  k = SHA256(s)[:16] = 6e340b9cffb37a989ca544e6bb780a2c

  [ok] Both shared secrets = 0 (as Mallory predicted)

Mallory knows s = 0:
  k_mallory = SHA256(0)[:16] = 6e340b9cffb37a989ca544e6bb780a2c
  [ok] Mallory's key matches Alice's and Bob's key

Alice -> Bob:
  m0 = Hi Bob!
  c0 = bb2f6bb3c984e38fc5d30528b938d8db

Bob -> Alice:
  m1 = Hi Alice!
  c1 = f4f526fcf0d564045533bbb3291c8f98

Mallory decrypts:
  c0 -> Hi Bob!
  c1 -> Hi Alice!
  [ok] Mallory successfully recovered both plaintext messages!

======================================================================
  Task 2 Part 2 -- Generator Tamper: a -> 1
======================================================================

Alice (using tampered a):
  X_A = 2901117983434351564088580898250638938537657764958169446838745126024418935322816838931396173560858936445354327587417253 2...
  Y_A = a^X_A mod q = 1

Bob (using tampered a):
  X_B = 5584269619193572666219681327859428721558625875869800798626680306856025612630728637362971159313673087980559991119676991 0...
  Y_B = a^X_B mod q = 1

Shared secret:
  Alice: s = 1
  Bob:   s = 1
  [ok] Shared secrets match: s = 1
  k = 4bf5122f344554c53bde2ebb8cd2b7e3

Alice -> Bob:  c0 = 443ab337885c88ecce7c27945ab42d27
Bob -> Alice:  c1 = 5d8d3a1b7ad949712e23f5a057700f69

Mallory's reasoning:
  a=1 ==> Y_A=Y_B=1 ==> s = 1^X mod q = 1
  Candidate shared secrets: [1]

  s = 1 -> k = 4bf5122f344554c53bde2ebb8cd2b7e3
    c0 decrypts to: Hi Bob!
    c1 decrypts to: Hi Alice!
    [ok] Mallory recovered both messages!

======================================================================
  Task 2 Part 2 -- Generator Tamper: a -> q
```

```
======================================================================

Alice (using tampered a):
  X_A = 32027198643683035952581244969726379315743599833659473661490043484934621996466959244986843299605412644141423123279197150...
  Y_A = a^X_A mod q = 0

Bob (using tampered a):
  X_B = 36109754442868628927043994241428058016581749859002553209968293442534770076541068116901701151886659722271642268662551882...
  Y_B = a^X_B mod q = 0

Shared secret:
  Alice: s = 0
  Bob:   s = 0
  [ok] Shared secrets match: s = 0
  k = 6e340b9cffb37a989ca544e6bb780a2c

Alice -> Bob:  c0 = bb2f6bb3c984e38fc5d30528b938d8db
Bob -> Alice:  c1 = f4f526fcf0d564045533bbb3291c8f98

Mallory's reasoning:
  a=q ==> Y_A=Y_B=0 ==> s = 0^X mod q = 0
  Candidate shared secrets: [0]

  s = 0 -> k = 6e340b9cffb37a989ca544e6bb780a2c
    c0 decrypts to: Hi Bob!
    c1 decrypts to: Hi Alice!
    [ok] Mallory recovered both messages!

======================================================================
  Task 2 Part 2 -- Generator Tamper: a -> q-1
======================================================================

Alice (using tampered a):
  X_A = 10615346490798077478064314263410130464494223125153014283844027908297658689704571056877916420665378501428462243720087163...
  Y_A = a^X_A mod q = 1

Bob (using tampered a):
  X_B = 99530176960339024730192101757904343974865446680246076731853606943569370343936839025334239713383254477431680345267888355...
  Y_B = a^X_B mod q = 1

Shared secret:
  Alice: s = 1
  Bob:   s = 1
  [ok] Shared secrets match: s = 1
  k = 4bf5122f344554c53bde2ebb8cd2b7e3

Alice -> Bob:  c0 = 443ab337885c88ecce7c27945ab42d27
Bob -> Alice:  c1 = 5d8d3a1b7ad949712e23f5a057700f69

Mallory's reasoning:
  a=q-1 ==> Y in {1, q-1} ==> s in {1, q-1}; Mallory tries both
  Candidate shared secrets: [1, 12432533914688938454049409108545663000985688274187280618173127901849182080011946002236740376979...

  s = 1 -> k = 4bf5122f344554c53bde2ebb8cd2b7e3
    c0 decrypts to: Hi Bob!
    c1 decrypts to: Hi Alice!
    [ok] Mallory recovered both messages!
```

Task 3: Textbook RSA & MITM via Malleability

Part 1 (Textbook RSA): Implements RSA key generation with variable-length primes and e=65537. The modular multiplicative inverse is computed using the Extended Euclidean Algorithm (implemented from scratch). Messages are converted from strings to integers and encrypted/decrypted successfully.

Part 2 (MITM via Malleability): Mallory intercepts Bob's ciphertext c and replaces it with c' = r^e mod n for a known r. When Alice decrypts c', she gets s' = r. Mallory knows r, derives the same AES key, and decrypts Alice's message.

Signature Malleability: Given signatures sig1 = m1^d mod n and sig2 = m2^d mod n, Mallory computes sig3 = sig1 * sig2 mod n, which is a valid signature for m3 = m1 * m2 mod n. This works because (m1^d)(m2^d) = (m1*m2)^d mod n.

Source Code: task3_rsa.py

```
#!/usr/bin/env python3
"""
Task 3: "Textbook" RSA & MITM Key Fixing via Malleability
==========================================================
Part 1 -- Textbook RSA: key generation, encryption, decryption
Part 2 -- MITM attack exploiting RSA malleability + signature forgery
"""

import secrets
from Crypto.Util.number import getPrime
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Util.Padding import pad, unpad


# ?? Helpers ??????????????????????????????????????????????????????????

def int_to_bytes(n: int) -> bytes:
    """Convert a non-negative integer to a big-endian byte string."""
    if n == 0:
        return b'\x00'
    length = (n.bit_length() + 7) // 8
    return n.to_bytes(length, byteorder='big')


def bytes_to_int(b: bytes) -> int:
    """Convert a big-endian byte string to an integer."""
    return int.from_bytes(b, byteorder='big')


def derive_key(value: int) -> bytes:
    """Derive a 16-byte AES key from an integer using SHA-256."""
    h = SHA256.new()
    h.update(int_to_bytes(value))
    return h.digest()[:16]


def aes_cbc_encrypt(key: bytes, iv: bytes, plaintext: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.encrypt(pad(plaintext, AES.block_size))


def aes_cbc_decrypt(key: bytes, iv: bytes, ciphertext: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size)


IV = b'\x00' * 16  # shared initialization vector


# =====================================================================
# Part 1 -- Textbook RSA Implementation
# =====================================================================
```

```python
def extended_gcd(a: int, b: int):
    """
    Extended Euclidean Algorithm.
    Returns (gcd, x, y) such that a*x + b*y = gcd(a, b).
    """
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y


def mod_inverse(e: int, phi: int) -> int:
    """
    Compute the modular multiplicative inverse of e modulo phi
    using the Extended Euclidean Algorithm.
    Returns d such that (e * d) mod phi = 1.
    Raises ValueError if inverse does not exist.
    """
    gcd, x, _ = extended_gcd(e % phi, phi)
    if gcd != 1:
        raise ValueError(f"Modular inverse does not exist (gcd={gcd})")
    return x % phi


def rsa_keygen(bits: int = 2048, e: int = 65537):
    """
    Generate an RSA key pair.

    Parameters
    ----------
    bits : int -- bit length of each prime (n will be ~2*bits bits)
    e    : int -- public exponent (default 65537)

    Returns
    -------
    (pub, pri) where pub = (n, e) and pri = (n, d)
    """
    while True:
        p = getPrime(bits)
        q = getPrime(bits)
        if p != q:
            break

    n = p * q
    phi = (p - 1) * (q - 1)  # Euler's totient

    # Verify gcd(e, phi) == 1  (almost always true for e=65537)
    g, _, _ = extended_gcd(e, phi)
    assert g == 1, "e and phi(n) are not coprime; regenerate primes"

    d = mod_inverse(e, phi)

    # Sanity check
    assert (e * d) % phi == 1, "Key generation error: e*d != 1 mod phi"

    return (n, e), (n, d)


def rsa_encrypt(pub: tuple, m: int) -> int:
    """Textbook RSA encryption: c = m^e mod n."""
    n, e = pub
    assert 0 <= m < n, "Message must be in Z*_n (i.e., 0 <= m < n)"
    return pow(m, e, n)


def rsa_decrypt(pri: tuple, c: int) -> int:
    """Textbook RSA decryption: m = c^d mod n."""
    n, d = pri
    return pow(c, d, n)
```

```
def demo_textbook_rsa():
    """Demonstrate textbook RSA key generation, encryption, and decryption."""
    print("=" * 70)
    print("  Task 3 Part 1 -- Textbook RSA")
    print("=" * 70)

    # Generate a 1024-bit key pair (each prime ~1024 bits -> n ~2048 bits)
    print("\nGenerating RSA key pair (1024-bit primes, e=65537)...")
    pub, pri = rsa_keygen(bits=1024)
    n, e = pub
    _, d = pri
    print(f"  n = {n}")
    print(f"  e = {e}")
    print(f"  d = {d}")
    print(f"  n bit-length = {n.bit_length()} bits")

    # Encrypt and decrypt a few messages
    messages = [
        "Hello, RSA!",
        "Textbook RSA is insecure.",
        "CSC321 Public Key Crypto",
    ]

    print(f"\n--- Encrypting and decrypting messages ---")
    for msg_str in messages:
        # Convert string -> bytes -> integer
        msg_bytes = msg_str.encode('utf-8')
        m = bytes_to_int(msg_bytes)
        assert m < n, "Message too large for this key!"

        c = rsa_encrypt(pub, m)
        m_dec = rsa_decrypt(pri, c)
        msg_dec = int_to_bytes(m_dec).decode('utf-8')

        print(f"\n  Plaintext:  \"{msg_str}\"")
        print(f"  m (int):    {m}")
        print(f"  Ciphertext: {c}")
        print(f"  Decrypted:  \"{msg_dec}\"")
        assert msg_dec == msg_str, "Decryption failed!"
        print(f"  [ok] Decryption successful")

    print()
    return pub, pri


# =====================================================================
# Part 2 -- MITM Attack via RSA Malleability
# =====================================================================

def demo_malleability_attack(pub, pri):
    """
    Demonstrate the MITM attack on textbook RSA key exchange.

    Protocol:
      1. Alice publishes (n, e)
      2. Bob picks random s in Z*_n, sends c = s^e mod n
      3. Mallory intercepts c, sends c' to Alice
      4. Alice decrypts s' = (c')^d mod n, derives k = SHA256(s')
      5. Alice encrypts m = "Hi Bob!" with AES-CBC_k and sends c0
      6. Mallory decrypts c0 using k (since she chose c' to know s')

    Attack: Mallory picks her own value r, computes c' = r^e mod n.
    When Alice decrypts c', she gets s' = r. Mallory knows r, so she
    can derive k = SHA256(r) and decrypt Alice's message.
    """
    print("=" * 70)
    print("  Task 3 Part 2 -- MITM Attack via RSA Malleability")
    print("=" * 70)

    n, e = pub
    _, d = pri
    # ?? Bob picks random s, computes c = s^e mod n ??
    s_bob = secrets.randbelow(n - 2) + 2  # s in Z*_n
    c = rsa_encrypt(pub, s_bob)
    print(f"\nBob:")
    print(f"  Picks random s = {s_bob}")
    print(f"  Sends c = s^e mod n = {c}")

    # ?? Mallory intercepts c and crafts c' ??
    # Mallory picks her own r and computes c' = r^e mod n
    # When Alice decrypts c', she gets s' = r
    r = secrets.randbelow(n - 2) + 2  # Mallory's chosen value
    c_prime = rsa_encrypt(pub, r)  # c' = r^e mod n
    print(f"\nMallory intercepts c and crafts c':")
    print(f"  Mallory picks r = {r}")
    print(f"  Mallory computes c' = r^e mod n = {c_prime}")
    print(f"  Mallory sends c' to Alice (instead of c)")

    # ?? Alice decrypts c' to get s' ??
    s_prime = rsa_decrypt(pri, c_prime)  # s' = (c')^d mod n = r
    k_alice = derive_key(s_prime)
    print(f"\nAlice:")
    print(f"  Decrypts c' -> s' = {s_prime}")
    print(f"  Derives k = SHA256(s')[:16] = {k_alice.hex()}")

    # Verify Alice recovered Mallory's r
    assert s_prime == r, "s' should equal r"
    print(f"  (s' == r: [ok])")

    # ?? Alice encrypts a message ??
    m = b"Hi Bob!"
    c0 = aes_cbc_encrypt(k_alice, IV, m)
    print(f"\n  Alice encrypts m = \"{m.decode()}\"")
    print(f"  c0 = {c0.hex()}")

    # ?? Mallory decrypts c0 ??
    # Mallory knows r, so she can compute k = SHA256(r)[:16]
    k_mallory = derive_key(r)
    m_mallory = aes_cbc_decrypt(k_mallory, IV, c0)
    print(f"\nMallory:")
    print(f"  Knows r = {r}")
    print(f"  Derives k = SHA256(r)[:16] = {k_mallory.hex()}")
    print(f"  Decrypts c0 -> \"{m_mallory.decode()}\"")
    assert m_mallory == m
    print(f"  [ok] Mallory successfully recovered the plaintext message!")

    print()


# ======================================================================
# Signature Malleability Demonstration
# ======================================================================

def demo_signature_malleability(pub, pri):
    """
    Demonstrate RSA signature malleability.

    Sign(m, d) = m^d mod n

    If Mallory sees signatures for m1 and m2:
      sig1 = m1^d mod n
      sig2 = m2^d mod n

    Then she can forge a signature for m3 = m1 * m2 mod n:
      sig3 = sig1 * sig2 mod n
           = (m1^d) * (m2^d) mod n
           = (m1 * m2)^d mod n
           = Sign(m1 * m2, d)
    """
    print("=" * 70)
    print("  Task 3 Part 2 -- RSA Signature Malleability")
    print("=" * 70)

    n, e = pub
    _, d = pri

    # Two messages (as integers)
    m1 = bytes_to_int(b"msg_one")
    m2 = bytes_to_int(b"msg_two")
    m3 = (m1 * m2) % n  # The forged message

    # Legitimate signatures
    sig1 = pow(m1, d, n)  # Sign(m1, d)
    sig2 = pow(m2, d, n)  # Sign(m2, d)

    print(f"\nLegitimate signatures:")
    print(f"  m1 = {m1}  (\"msg_one\")")
    print(f"  m2 = {m2}  (\"msg_two\")")
    print(f"  sig1 = m1^d mod n = {sig1}")
    print(f"  sig2 = m2^d mod n = {sig2}")

    # Verify legitimate signatures
    assert pow(sig1, e, n) == m1, "sig1 verification failed"
    assert pow(sig2, e, n) == m2, "sig2 verification failed"
    print(f"\n  [ok] sig1 verifies: sig1^e mod n == m1")
    print(f"  [ok] sig2 verifies: sig2^e mod n == m2")

    # Mallory forges signature for m3 = m1 * m2 mod n
    sig3_forged = (sig1 * sig2) % n
    print(f"\nMallory forges signature for m3 = m1 * m2 mod n:")
    print(f"  m3 = {m3}")
    print(f"  sig3 = sig1 * sig2 mod n = {sig3_forged}")

    # Verify forged signature
    verified = pow(sig3_forged, e, n)
    print(f"\nVerification:")
    print(f"  sig3^e mod n = {verified}")
    print(f"  m3           = {m3}")
    assert verified == m3, "Forged signature verification failed!"
    print(f"  [ok] Forged signature is valid! sig3^e mod n == m3")

    # Compute legitimate signature for m3 to cross-check
    sig3_legit = pow(m3, d, n)
    assert sig3_forged == sig3_legit, "Forged sig doesn't match legitimate sig"
    print(f"  [ok] Forged signature matches the legitimate signature for m3")

    print()


# ======================================================================
# Main
# ======================================================================

if __name__ == "__main__":

    # Part 1: Textbook RSA
    pub, pri = demo_textbook_rsa()

    # Part 2: MITM attack via malleability
    demo_malleability_attack(pub, pri)

    # Part 2 (continued): Signature malleability
    demo_signature_malleability(pub, pri)
```

Output

```
======================================================================
  Task 3 Part 1 -- Textbook RSA
======================================================================

Generating RSA key pair (1024-bit primes, e=65537)...
  n = 1948134936300534689284179059699430264422633752382969463192948280019054663327625849591784456669725163398821294810386068196...
  e = 65537
  d = 1377757696607923498384613504402229173226187524439602844950637957651451311329052756499833953067505250444359189361224710847...
  n bit-length = 2048 bits

--- Encrypting and decrypting messages ---

  Plaintext:  "Hello, RSA!"
  m (int):    87521618088895491219865889
  Ciphertext: 1289045616926262715319034988793111502941697959130536686636787336114346681391545627016660687090881519794093609 9448...
  Decrypted:  "Hello, RSA!"
  [ok] Decryption successful

  Plaintext:  "Textbook RSA is insecure."
  m (int):    529764595827692920050964175412001605351535649826477433840942
  Ciphertext: 1931275864176523017288502627467366407057407923845725040789340143977128366208479391885839893939584807834716010 4013...
  Decrypted:  "Textbook RSA is insecure."
  [ok] Decryption successful

  Plaintext:  "CSC321 Public Key Crypto"
  m (int):    1650810182925686748163473662496056991329488056922177303663
  Ciphertext: 1793974861233556359644077646462848755354637906119910122953458338337520597795106216741692740482767257719786785 8420...
  Decrypted:  "CSC321 Public Key Crypto"
  [ok] Decryption successful

======================================================================
  Task 3 Part 2 -- MITM Attack via RSA Malleability
======================================================================

Bob:
  Picks random s = 9506514874785467529189288174081278576608429376803820043158442411632468643980140656986582635693162904838881 95...
  Sends c = s^e mod n = 6773288364436212971025295673730557509751343187035816130952472488905026835008132866156170039661148228852...

Mallory intercepts c and crafts c':
  Mallory picks r = 1883211312913099143094930467425651466437248191925231556795992816248958405480854857494201207093651661075550 4...
  Mallory computes c' = r^e mod n = 2056145226651338272609242753238494993363055389572309003415783504511930101001532563442799897...
  Mallory sends c' to Alice (instead of c)

Alice:
  Decrypts c' -> s' = 188321131291309914309493046742565146643724819192523155679599281624895840548085485749420120709365166107555...
  Derives k = SHA256(s')[:16] = 97107486578c67bec16ab31f13c05bb3
  (s' == r: [ok])

  Alice encrypts m = "Hi Bob!"
  c0 = e4e007864d7ebb5a84c4857d5d958ce8

Mallory:
  Knows r = 18832113129130991430949304674256514664372481919252315567959928162489584054808548574942012070936516610755504010181 58...
  Derives k = SHA256(r)[:16] = 97107486578c67bec16ab31f13c05bb3
  Decrypts c0 -> "Hi Bob!"
  [ok] Mallory successfully recovered the plaintext message!

======================================================================
  Task 3 Part 2 -- RSA Signature Malleability
======================================================================

Legitimate signatures:
  m1 = 30807660281425509  ("msg_one")
  m2 = 30807660281755503  ("msg_two")
  sig1 = m1^d mod n = 1354835538340165676346068372702890650264696803358595536184437603667366928176550753979700100422941361791 71...
  sig2 = m2^d mod n = 1569919550312570675094112936871864987867512133512116307037943363787699895517848508727361590242930398396 11...

  [ok] sig1 verifies: sig1^e mod n == m1
  [ok] sig2 verifies: sig2^e mod n == m2

Mallory forges signature for m3 = m1 * m2 mod n:
  m3 = 949111932025889215445105845326027
  sig3 = sig1 * sig2 mod n = 1300656728329279569085494599638198534938930333522611249635904793881282685912732879839096246636802 4...

Verification:
```

```
sig3^e mod n = 949111932025889215445105845326027
m3           = 949111932025889215445105845326027
[ok] Forged signature is valid! sig3^e mod n == m3
[ok] Forged signature matches the legitimate signature for m3
```